Error Containment

Robert S. Hanmer
Alcatel-Lucent
2000 Lucent Lane 2H-207
Naperville, IL 60566-7033
hanmer@alcatel-lucent.com

Abstract:

High availability is required in many computer systems today. These include web servers, e-commerce applications, network and telephony devices such as routers and switches, and many others. No software is defect free, and neither is the environment in which the software operates. As a result errors happen. To prevent errors from causing failures they must be kept from spreading. The goal is to limit the parts of the system that the error *infects* with its incorrectness. These patterns discuss two ways of containing errors.

The patterns in this paper describe ways to limit error propagation through the containment of errors. Error containment is an essential part of error detection and mitigation. The objective is to tolerate faults and errors that exist in the system to allow general system operation to continue. An aspect of tolerating faults is to limit their effects. Errors in one part of the system should not cause errors in other parts of the system. Once an error is detected the system must either *recover* from the error by moving the system to a state that does not contain the error, or *mitigate* the error by undoing the error to put the system in a state equivalent to the current one but without the error. *Containment*, the subject of these patterns, is focused on preventing the current error state from spreading.

The terms *fault, error* and *failure* have specific meanings.

> A system **failure** occurs when the delivered service no longer complies with the **specification**, the latter being an agreed description of the system's expected function and/or service. An **error** is that part of the system state that is liable to lead to subsequent failure; an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a **fault**. [Lap91]

A *fault* is the defect that is present in the system that can cause an error. The fault might be a latent software "bug", or it might be a garbled message received on a communications channel, or a variety of other things. In general, software is not aware of the presence of a fault until an *error* occurs. An example of a software fault is a misplaced decimal point in a data constant, for example the number of steps needed to rotate an assembly robot's arm one degree.

An *error* is the manifestation of the fault, usually an incorrect action taken by the system. Continuing the example, the incorrect result of an arithmetic computation made

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

with the faulty data constant is the error; for example the number of steps an order of magnitude too large for a certain desired motion because of the misplaced decimal point.

The *failure* is the deviation from the agreed-upon correct operation of the system. In the case of the robots arm, the failure might be that it rotates in the wrong direction because of the erroneous computation made with the faulty decimal point.

The four phases of fault tolerance: *error detection, error recovery, error mitigation* and *fault treatment* describe the execution time lifecycle of a fault that is present in a system. Assuming that there is a latent fault in the system, at first it must be detected. This can happen through a routine means such as an audit (checksum) check, or it might be detected when an error is detected. The presence of the error is an indicator that there is a fault present in the system. The fault is no longer latent but is active when an error occurs. A number of patterns that discuss detection are available: for example Watchdog Detection [Han04], "A system of Patterns for Fault Tolerance" by Titos Saridakis [Sar02], and also many examples in Patterns for Time-Triggered Embedded Systems by Michael Pont [Pon01].

Once the error is detected it must be processed. Depending on the type of error, the processing either consists of *error recovery* or *error mitigation*. Both of these types of processing work to contain the error to prevent its effects from spreading to other parts of the system causing other errors, or even a failure. Error recovery moves the system to a state that does not contain the error. ROLLBACK and ROLL-FORWARD [Lea00] are two patterns that support error recovery. Error mitigation masks the error without changing the system's execution state (meaning the program counter doesn't change). CORRECTING AUDITS and ERROR CORRECTING CODES (both unwritten) are examples of error mitigation patterns.

Fault treatment is done last, and is the step in which the fault is removed from the system through a process of diagnosis and correction.

Another phase, fault prevention reflects the ability during design and development to avoid the insertion of the fault into the system. Fault prevention is performed at design time, not at execution time.

The first pattern in this paper, ERROR CONTAINMENT BARRIER (1)[1], is a pattern to assist in the error detection phase. The second pattern, MARKED DATA (2), is an error mitigation pattern that describes a way of mitigating erroneous data that is detected during execution.

Some assumptions must be outlined to understand the larger context of these patterns. These assumptions represent things that are covered by other portions of the larger work that includes these patterns.

[1] Pattern names will be presented in ALL CAPS. Pattern names are usually followed either by an internal reference number contained within parenthesis, or by a reference to a published paper. Thumbnails for all the patterns mentioned appear at the end.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

- Reporting to the fault tolerance control entity (who might be human) about all errors detected and actions taken is a basic function that all parts of the system must do.
- Most of the patterns are narrowly focused so that they are small enough that an individual developer or a small team can include the artifacts that they describe in their designs. As a result there are these following assumptions:
    - The basic framework to support fault tolerance is in place in the system.
    - The Fault Observer receives reports, but does not micro-manage the actions of the objects discussed in these patterns.
    - To simplify talking about concepts the activities that are described might be best integrated with other functionality (either application or fault tolerance related). Patterns are used to explain the activities that must be designed; the patterns do not constrain the implementation of the functionality.
- The system capabilities that are discussed in these patterns rest on top of the application-required functionality and are orthogonal to it in many ways.
- Achieving fault tolerance and maintaining a state of fault tolerance are not free. Both development and execution resources are required.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

1. ERROR CONTAINMENT BARRIER

… The system is designed to perform as well as it can in the presence of faults. The necessary fault tolerance framework elements are in place. The software knows that it is supposed to be within a highly available system.

There are mechanisms for detection of faults that have been designed into the system sprinkled throughout the various UNITS OF MITIGATION (unwritten).

Errors in one part of the system, or in one computation, can spread and cause errors or failures in other parts of the system. Errors spread through several mechanisms: erroneous messages, corrupted (incorrect) pooled memory or actions based on the results of other incorrect actions.

The error(s) have been detected through some detection mechanisms such as WATCHDOGS, SYSTEM MONITORS, ROUTINE AUDITS and others (all unwritten).

❖ ❖ ❖

How do you contain an error and keep it from propagating?

Unless something is done the error will continue through the system forever or until it eventually causes a failure that results in termination. This is the nature of errors.

The effects of an error cannot always be predicted in advance. Nor can all the potential errors be predicted. Software reliability modeling, verification techniques and software quality efforts reduce the number of faults present in the system. The system must be adaptable and able to handle unanticipated errors. Any capability that the software has to deal with the effects of an error must be put in place during the design phase. The capabilities require conscious preparation.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

Fault tolerance is living with faults. In order to live with faults present in the system is to find a way to ignore or mask them. But some ways of masking errors result in their still being propagated throughout the system. If the system doesn't just "ignore" errors, what can it do? One option is to say "HELP" and terminate. But this does not fit into the framework of fault tolerance within the system (see MINIMIZE HUMAN INTERVENTION [ACG+96]). Sometimes terminating is the only practical option, for example when an error is detected that makes the system unsafe.

Another option is to take steps to mitigate the error. This isn't always possible though; it depends on the nature of the error and the fault. Some errors, particularly data errors can be mitigated by means such as CORRECTIVE AUDITS (unwritten)

In the case of some errors an effective way of mitigation is to mark them for all other parts of the system to know that they are erroneous. This eliminates the need for the other parts of the systems to detect the errors; they can concentrate on taking the steps appropriate for them to mitigate them. This is discussed in MARKED DATA (2).

Sometimes the errors cannot be corrected, and moreover stopping the system is not possible because we choose to MINIMIZE HUMAN INTERVENTION is being used so simply stopping is not sufficient. In these cases error recovery steps must be taken to move the system to an error free state.

Therefore,

Stop the flow of errors from one part to another by either initiating error recovery or error mitigation. Stop the error from progressing and then invoke appropriate notification, logging, mitigation and recovery functions. Do not leave the error unprocessed.

These techniques can be used to contain the error:

1. Mitigate the error, either by
   a. Marking the erroneous data for avoidance, as described in MARKED DATA (2), and define rules for processing erroneous data; or by
   b. Masking the erroneous element so that it is no longer erroneous, for example CORRECTING AUDITS.
2. Initiate a recovery action TO put the system in an error-free state. CHECKPOINT [Han03], ROLLBACK [Lea00] or ROLL-FORWARD [Lea00] discuss several recovery options.

Regardless of the technique used, report the error to the appropriate fault handlers (both internal and via FAULT OBSERVERS (unwritten)) for higher-level mitigation.

In order to be able to contain errors the system must be able to detect them. Additionally it must have the ability to decide what the course of action is the safest given the circumstances of the error. Detection as close to the fault in either structural proximity or time is the best-case scenarios.

❖ ❖ ❖

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

Hardware error containment can include isolating faulty hardware components through activity bits and other techniques.

MARKED DATA (2) describes a method of marking erroneous information to contain its future use.

ROLLBACK and ROLL-FORWARD discuss ways of transitioning to an error-free state.

"Design Patterns for Fault Containment" by Titos Saridakis [Sar03] contains three patterns that deal with guarding against errors propagating. Two of the patterns describe the use ways to detect and contain the spread of errors through an INPUT GUARD or an OUTPUT GUARD. The third pattern describes a CONTAINER object. …

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

2. MARKED DATA

… The system has a way to detect errors in data that it uses. Once detected, an ERROR CONTAINMENT BARRIER (1) will initiate processing of the error.

Erroneous data was detected, either in a message that is passing through this part of the system, or in an element of data that was stored earlier and accessed by this part of the system. Alternatively, the error might be detected as the result of an operation.

The system doesn't have enough information to be able to correct the erroneous data automatically. This can be because there isn't any record or a priori knowledge of what the correct value should be, or it does not contain sufficient information embedded within it to be corrected, e.g. it does not contain any ERROR CORRECTING CODES (unwritten).

The error that has been detected has a limited scope that doesn't require that the system state be greatly altered even though it can't be immediately corrected. In other words, error mitigation is more appropriate than error recovery actions such as restoring from a CHECKPOINT [Han03], or conducting a ROLLBACK and ROLL-FORWARD [Lea00].

❖ ❖ ❖

When uncorrectable erroneous data is found, how can the error be kept from spreading?

Sometimes stored data contains an error, for example when it is something that was put away for later use into a medium to longer-term storage. The part of the system that is going to contain the error might not have enough information to be able to determine if it was incorrect when first stored or if it was corrupted during storage. Using the invalid data will cause a failure; it must be contained to prevent this from happening. The corruption may have occurred in the past, but it remained unidentified until the data is about to be used. ROUTINE AUDITS (unwritten) are used to detect corrupt data before the data is needed for processing. In many cases CORRECTING AUDITS can be written to correct these elements of faulty data. But if the audits aren't available or the nature of the data prevents automatic correction then the data won't be correctable.

The storage medium can be made to tolerate errors on its own. For example the memory of the system can be designed to contain ERROR-CORRECTING CODES. These codes can only detect a certain number of bit errors in a given memory unit, but this will be sufficient for many error cases. This memory is common in systems that are designed from the hardware-up to be fault tolerant, but these error correcting and detecting code memories add expense.

If the data cannot be corrected it must be contained. In the short term the entity that detects that it is erroneous should not use it. The results of any actions taken with that data can be discarded.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

We also don't want the data to be used by any other parts of the system. It can be marked in a way that other parts of the system don't have to spend much time detecting that it was erroneous, and can quietly contain the impact of the error. Rules for how to proceed with data items that has been marked as erroneous are encountered must be defined.

In some cases merely marking the data as erroneous is insufficient and active error recovery steps need to be taken to contain an error. For example when there is no correct action possible because the erroneous data is to be used to control a branching of program execution.

The IEEE "Not a Number" is an example for marking a value erroneous in a way that allows processing to continue. The IEEE standard 754-1985 defines standard representations for binary floating-point numbers. While defining the numerical representation they also define a special value "Not a Number" or "NaN". NaN is stored in place of a floating-point value as the result of certain illegal floating-point operations, for example division by zero. The standard defines rules for how subsequent computations should behave when one of the operands is NaN. [IEEE754]

Rules for processing an operand that was marked by someone else as being erroneous should include two different types of information, both present in the NaN rules from the IEEE. The first type of information is how the operation should proceed. Possible rules include assuming a default value, skipping the operation and marking the result as erroneous, seeking the information from an alternate source, aborting execution or taking an EXCEPTION (unwritten) and so on. The second type of information that should be part of any rule is whether any notification to other parts of the system should be made previously marked erroneous value is encountered. The IEEE standard refers to this as signaled or quiet. This signaling is appropriate if some intermediate mechanism would have been expected to correct the error and so the current occurrence of the erroneous flag is totally unexpected.

Messages sometimes contain data elements that are erroneous. These must also be contained. In some cases the entire message can be discarded. This is most effortless when the protocol supports retransmission until received and the message has not been acknowledged yet.

Individual data elements within the message are sometimes identifiable as being erroneous. If only parts of a message are incorrect then a mechanism such as the IEEE NaN can be used to identify the erroneous part. This allows computing to continue while taking into account the elements that are erroneous.

When the results of a computation or processing are determined to be erroneous the NaN approach of marking the data element can work as well. In some cases the detection of an error at this level indicates that the part of the system that performed the computation is erroneous. In these cases the entire part of the system should be marked and avoided rather than just the result. In these cases a marking is needed, but the IEEE NaN is too low level. One approach is to report to the FAULT OBSERVER (unwritten) and rely on higher-level system functions to contain and repair the faulty entity.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

Marking data or results so that they aren’t used isn’t free. In the case of IEEE NaN the mark is encoded in place of the value, but sometimes the NaN mark might require additional “meta-memory”. Resources are required to check for the erroneous mark and take appropriate actions.

Therefore,

Mark erroneous data to indicate that it should not be used. Define rules for how these values should be processed when they are encountered.

The IEEE defines two types of NaNs, “signaling” and “quiet” and rules for how the implementations should handle these types of NaNs.

❖ ❖ ❖

The periodic checking of data for correctness is a variant of this pattern. Instead of waiting for the data to be accessed in normal operations, the ROUTINE AUDIT mechanism will periodically check for correctness and either mark or correct it.

“CHECKS” by Ward Cunningham [Cun95] introduces the idea of an exceptional value as a computational result. This effectively contains the error to everywhere upstream from where it is detected. Failures are prevented because the system does not use the erroneous value if it is flagged as exceptional. …

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

Pattern Thumbnails

Pattern	Intent
CHECKPOINT	Save state periodically in a way that allows execution to be resumed with a consistent state.
CONTAINER	Perform computations in a safe space that is being protected by INPUT GUARDS and OUTPUT GUARDS.
CORRECTING AUDIT	Some data can be corrected based on other data found in the system. Correct it and return it to usefulness
ERROR CONTAINMENT BARRIER (1)	Build barriers into your system so that errors can't propagate from one part of the system to another.
ERROR CORRECTING CODES	Store enough extra information with each data value that will allow it to be corrected for bit errors in a storage device.
FAULT OBSERVER	Some part of the system should know that a fault is present and report it and maybe escalate actions.
INPUT GUARD	Detect errors and prevent them from spreading by checking input values.
MARKED DATA (2)	Mark erroneous data values as invalid and define rules for how to process these values when encountered later to prevent any part of the system from propagating the error.
MINIMIZE HUMAN INTERVENTION	Humans make mistakes and are slow; to minimize downtime the system should take care of itself, without human intervention.
OUTPUT GUARD	Detect errors on exit from a module and prevent them from spreading.
ROLLBACK	Move the system to an error-free state by returning to a state before the error occurred.
ROLL-FORWARD	Move the system to an error-free state by advancing to a future state that doesn't contain the error.
ROUTINE AUDIT	Check data with a background task to make sure that it is correct
UNITS OF MITIGATION	Decide during architecture what the units of fault tolerance are.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.

Acknowledgements:

Thanks to Dirk Riehle who shepherded this paper through many changes of direction. Ralph Johnson provided valuable feedback on the patterns.

The photo accompanying Error Containment Boundary is used courtesy of the US National Archives and Records Administration. *www.nara.gov* ARC identifier 198978.

A large thank you to the PLoP 2006 Writers' Workshop Group *Intimacy Gradient* for their comments and suggestions on this paper. The group consisted of: Mirko Raner, Erol Thompson, Anders Janmyr, Philipp Bachmann, Daniel Vainsencher, Maurice Rabb, Andrew Black, Sachin Bammi, Kanwardeep Singh Ahluwalia, Ward Cunningham, Brian Foote, Ademar Aguiar and Ricardo Lopez.

References

[ACG+96] Adams, M. E., J. O. Coplien, R. J. Gamoke, R. S. Hanmer, F. Keeve, and K. L. Nicodemus. "Fault-Tolerant Telecommunications System Patterns." In [VCK96], pp 549-573.

[Cun95] Cunningham, W., "The CHECKS Pattern Language of Information Integrity." In [CS95], pp 145-155.

[CS95] Coplien, J. and Schmidt, D., eds. **Pattern Languages of Program Design**. Reading: Addison-Wesley, 1995.

[Han03] Hanmer, R. S., "Patterns of System Checkpointing," in Proceedings of 2003 PLoP Conference.

[Han04] Hanmer, R. S., "Watchdog Detection," in Proceedings of 2004 PLoP Conference.

[IEEE754] -. **IEEE 754-1984, IEEE Standard for Binary Floating-Point Arithmetic**. New York: IEEE 1985.

[Lap91] Laprie, J. C. **Dependability: Basic Concepts and Terminology**. New York: Springer-Verlag, 1991, p 4.

[Lea90] Lea, D. **Concurrent Programming in Java, Second Edition: Design Principles and Patterns.**. Reading, MA: Addison-Wesley, 2000.

[Pon01] Pont, M. J. **Patterns for Time-Triggered Embedded Systems**. New York, ACM Press, 2001.

[Sar02] Saridakis, T., "A System of Patterns for Fault Tolerance," in Proceedings of 2002 EuroPLoP Conference.

[Sar03] Saridakis, T., "Design Patterns for Fault Containment," in Proceedings of 2003 EuroPLoP Conference.

[VCK96] Vlissides, J., J. Coplien and N. Kerth, eds. **Pattern Languages of Program Design-2**. Reading, Mass: Addison-Wesley, 1996.

Copyright © 2006, Lucent Technologies. All Rights Reserved.
Permission is granted to copy for the PLoP 2006 conference.